CSE 5525 Homework #3:
Semantic Parsing with Encoder-Decoder Models

Deadline: **11:59PM on 11/05/2021**

Academic Integrity

You are free to discuss the homework assignments with other students and study course materials together towards solutions. **However, all of the code and report you write must be your own!** If you do discuss or study together with others (except via posts/comments on Teams), please list their names at the top of your written submission. Unless there is academic misconduct detected, this won't be counted against your credit.

Plagiarism will be automatically detected on Carmen, e.g., by comparing your HW with all resources, as well as manually checked by TA. If the instructor or TA suspects that a student has committed academic misconduct in this course, they are obligated by university rules to report their suspicions to the Committee on Academic Misconduct. Please see more in Statements for CSE5525 on Carmen pages.

Goals

In this project you'll implement an encoder-decoder model for semantic parsing. A sample encoder implementation in Pytorch is provided to you. You will have to figure out how to implement the decoder module, combine it with the encoder, do training, and do inference. Additionally, you'll be exploring attention mechanisms in encoder-decoder models.

The expectation is that you understand the encoder-decoder models with attention mechanisms from scatch, but you are allowed to implement them based on Pytorch. **You can call existing functions in Pytorch, as long as the required steps are shown.**

Timeline & Credit

You will have around 4 weeks (after the HW2 due date) to work on this programming assignment. Late submissions will NOT be accepted. So, start early and plan ahead! We currently use a 100-point scale for this homework (Part 1: 50 points; part 2: 30 points; 1-2 page report: 20 points), but it will take 20% of your final grade.

Questions?

Please create a post on MS Teams to get timely help from other students, the TA, and the instructor. Remember that participation takes 5% of your final grade. You can show your participation by actively answering others' questions and online class participation! **Besides, everyone can benefit from checking what has been asked previously.** Feel free to join the TA or the instructor's office hour as well (Time and Zoom links can be found on Carmen).

1 Background and Dataset

Semantic parsing involves translating natural language (NL) utterances (e.g., question) into various kinds of logical forms (LF) or formal representations such as SQL query, lambda calculus or lambda-DCS. These representations' main feature is that they fully disambiguate the natural language and can

effectively be treated like source code: executed to compute a result in the context of an environment such as a knowledge base. In this case, you will be dealing with the Geoquery dataset (Zelle and Mooney, 1996; [3]). Two examples (i.e., NL questions and their corresponding logical forms) from this dataset formatted as you'll be using are shown below:

(1) NL Question: `what is the population of atlanta ga ?`
LF: `_answer ( A , ( _population ( B , A ) , _const ( B , _cityid ( atlanta , _ ) ) ) )`

(2) NL Question: `what states border texas ?`
LF: `_answer ( A , ( _state ( A ) , _next_to ( A , B ) , _const ( B , _stateid ( texas ) ) ) )`

These are Prolog formulas similar to the lambda calculus expressions we have seen in class. In each case, an answer is computed by executing this expression against the knowledge base and finding the entity `A` for which the expression evaluates to true. You will be following in the vein of Jia and Liang (2016) [1], who tackle this problem with sequence-to-sequence (seq2seq) models. These models are not guaranteed to produce valid logical forms, but circumvent the need to come up with an explicit grammar, lexicon, and parsing model. In practice, encoder-decoder models can learn simple structural constraints such as parenthesis balancing (when appropriately trained), and typically make errors that reflect a misunderstanding of the underlying sentence, i.e., producing a valid but incorrect logical form, or "hallucinating" things that weren't there. We can evaluate these models in a few ways: (1) based on the denotation (the answer that the logical form gives when executed against the knowledge base), (2) based on simple token-level comparison against the reference logical form, and (3) by exact match against the reference logical form (slightly more stringent than denotation match). **For background on Pytorch implementations of seq2seq models, check out the helpful tutorial at this URL:** `https://pytorch.org/tutorials/intermediate/seq2seq_translation_tutorial.html`.

1.1 Dataset

The data consists of a sequence of (NL sentence, logical form) pairs. `geo_train.tsv` contains a training set of 480 pairs, `geo_dev.tsv` contains a dev set of 120 pairs, and `geo_test.tsv` contains a blind test set of 280 pairs (the standard test set). The test file has been filled with junk logical forms (a single one replicated over each line), so it can be read and handled in the same format as the others.

1.2 Programming Setup

This assignment will assume you are using Pytorch, Python 3, and numpy. The framework code provided to you will be in Pytorch, so it's recommended that you familiarize yourself with Pytorch. However, if you want to, you're free to use Tensorflow for this project, though you will need to implement the encoder yourself (The encoder in the given framework code is in Pytorch). You will also need Java for executing the evaluator for logical forms on Geoquery.

How to install. In addition to the setup for previous assignments, you will need to install Pytorch and its dependencies. You should follow the instructions at `https://pytorch.org/get-started/locally/`. **All assignments for this course are small-scale enough to complete using CPUs, so don't worry about installing CUDA and getting GPU support working unless you want to.**

Installing in a virtual environment is usually best; we recommend using anaconda, especially if you are on OS X, where the system python has some weird package versions. Once you have anaconda installed, you can create a virtual environment and install pytorch with the following commands:

```
conda create -n my-cse5525-virtenv python=3
conda install -n my-cse5525-virtenv -c pytorch pytorch torchvision
```

where `my-cse5525-virtenv` can be any name you choose. (You can also install tensorflow, tensorboard, or any other necessary packages in this virtual environment.)

1.3 Framework Code

We provide several code pieces for you to start with:

`main.py`: Main framework for argument parsing, setting up the data, training, and evaluating models. It contains a `Seq2SeqSemanticParser` class: this is the product of training and you will implement its

`decode` function, as well as the `train_model_encdec` function to train it. It also contains an `evaluate` function that evaluates your model's output.

`models.py`: Contains an implementation of an encoder. This is a Pytorch module that consumes a sentence (or batch of sentences) and produces (h, c) vector pairs. This is a vetted implementation of an LSTM that is provided for your convenience; however, if you want to build something yourself or use some open-sourced encoder implementations (Please give citations, if you use others'), you should feel free! Note that this implementation does not use GloVe embeddings, but you're free to use them if you want; you just need to modify the embedding layer. See the documentation for `encode_input_for_decoder` in `main.py`, which describes its usage.

`data.py`: Contains an `Example` object which wraps a pair of sentence or question (x) and logical form (y), as well as tokenized and indexed copies of each. `load_datasets` loads in the datasets as strings and does some necessary preprocessing. `index_datasets` then indexes input and output tokens appropriately, adding an `EOS` token to the end of each output string.

`lf_evaluator.py`: Contains code for evaluating logical forms and comparing them to the expected denotations. This calls a backend written in Java by Jia and Liang (2016) [1]. You should not need to look at this file, unless for some reason you are getting crashes during evaluation and need to figure out why the Java command is breaking.

`utils.py`: Same as before.

Next, try running
`python main.py --do_nearest_neighbor`
This runs a simple semantic parser based on nearest neighbors: return the logical form for the most similar example in the training set. This should report a denotation accuracy of 24/120 (it's actually getting some examples right!), and it should have good token-level accuracy as well. You can check that the system is able to access the backend without error.

2 What You Need to Do

2.1 Part 1: Basic Encoder-Decoder (50 points)

Your first task is to implement the basic encoder-decoder model. There are three things you need to implement.

Model You should implement a decoder module in Pytorch. One good choice for this is a single cell of an LSTM whose output is passed to a feedforward layer and a softmax over the vocabulary. You can piggyback off of the encoder to see how to set up and initialize this, though not all pieces of that code will be necessary. This cell should take a single token and a hidden state as input and produce an output and a new hidden state. At both training and inference time, the input to the first decoder cell should be the output of the encoder.

Training You'll need to write the training loop in `train_model_encdec`. Parts of this have been given to you already. You should iterate through examples, call the encoder, scroll through outputs with the decoder, accumulate log loss terms from the prediction at each point, then take your optimizer step. You probably want to use "teacher forcing" where you feed in the correct token from the previous timestep regardless of what the model does. Training should return a `Seq2SeqSemanticParser`. You will need to expand the constructor of this method to take whatever arguments you need for decoding: this probably includes one or more Pytorch modules for the model as well as any hyperparameters.

Inference You should implement the `decode` method of `Seq2SeqSemanticParser`. You're given all examples at once in case you want to do batch processing. This looks somewhat similar to the inner loop of training: you should encode each example, then repeatedly call the decoder. However, in this case, you want the most likely token out of the decoder at each step until the stop token (i.e., EOS) is generated. Then, de-index these and form the Derivation object as required.

Expectation. After 10 epochs taking 50 seconds per epoch, the reference implementation can get roughly 70% token accuracy and 10% denotation accuracy. You can definitely do better than this with larger models and training for longer, but attention is necessary to get much higher performance. **Your LSTM should be in roughly this ballpark; you should empirically demonstrate that it "works," but there is no hard minimum performance.**

2.2 Part 2: Attention (30 points)

Your model likely does not perform well yet; even learning to overfit the training set is challenging. One particularly frustrating error it may make is predicting the right logical form but using the wrong constant, e.g., always using `texas` as the state instead of whatever was said in the input. Attention mechanisms are a major modification to sequence-to-sequence models that are very useful for most translation-like tasks, making models more powerful and faster to train.

Attention requires modifying your model as described in lecture: you should take the output of your decoder RNN, use it to compute a distribution over the input's RNN states, take a weighted sum of those, and feed that into the final softmax layer in addition to the hidden state. This requires passing in each word's representation from the encoder, but this is available to you as `output` (returned by the encoder).

You'll find that there are a few choice points as you implement attention. First is the type of attention: linear, dot product, or general, as described in Luong et al. (2015) [2]. Second is how to incorporate it: you can compute attention before the RNN cell (using h_{t-1} and x) and feed the result in as (part of) the cell's input, or you can compute it after the RNN cell (using h_t) and use it as the input to the final linear and softmax layers. Feel free to play around with these decisions and others!

Expectation. After only 10 epochs taking 20 seconds per epoch, using Luong style "general" attention [2] can get roughly 77% token accuracy and 30-45% denotation accuracy (it's highly variable), achieving 80% token / 53% denotation after 30 epochs. **To get full credit on this part, your LSTM should get at least 50% denotation accuracy on some training run.**

Implementation and Debugging Tips:

- One common sanity check test for a sequence-to-sequence model with attention is the copy task: try to produce an output that's exactly the same as the input. You can train your model by changing the output of each example to be the same as your input, i.e., using (sentence, sentence) pairs rather than (sentence, logical form) pairs. Your model should be able to learn this copy task *perfectly* after just a few iterations of training. If your model struggles to learn this or tops out at an accuracy below 100%, there's probably something wrong.
- Optimization in sequence-to-sequence models is tricky! Many optimizers can work. For SGD, one rule of thumb is to set the step size as high as possible without getting `NaNs` in your network, then decrease it once performance on validation set stops increasing. For Adam, step sizes of 0.01 to 0.0001 are typical when you use the default momentum parameters, and higher learning rates can often result in faster training.
- If using dropout, be sure to toggle `module.train()` on each module before training and `module.eval()` before evaluation.
- Make sure that you do everything in terms of Pytorch tensors! If you do something like take a Pytorch tensor and convert to numbers and back, Pytorch won't be able to figure out how to do backpropagation.

3 What You Should Submit and Grading Expectation

You should submit the following to Carmen:

- Your code as a .zip or .tgz file
- Your model's output on the blind test set.
- A report of around 1-2 pages in PDF (20 points), not including references, though you aren't expected to reference many papers. Your report should list your collaborators (i.e., who you discussed HW#3 with, if any; see the beginning of this pdf), **briefly** restate the core problem and what you are doing, describe relevant details of your implementation, present results on the development set from Part 1 and Part 2, and briefly discuss a few error cases in both parts and how the system could be further improved.

Grading Expectation. Your assignment is judged roughly 80% based on your implementation (50% for part 1 and 30% for Part 2) and 20% on the report. **Even if you cannot make your implementation work, describe your effort.**

3.1 Acknowledgment

This homework assignment is based on NLP courses by Dr. Greg Durrett at UT Austin.

References

[1] Robin Jia and Percy Liang. Data recombination for neural semantic parsing. *arXiv preprint arXiv:1606.03622*, 2016.

[2] Minh-Thang Luong, Hieu Pham, and Christopher D Manning. Effective approaches to attention-based neural machine translation. *arXiv preprint arXiv:1508.04025*, 2015.

[3] John M Zelle and Raymond J Mooney. Learning to parse database queries using inductive logic programming. In *Proceedings of the national conference on artificial intelligence*, pages 1050–1055, 1996.